(2)

FINAL REPORT

Graphics-Based Parallel Programming Tools

Janice E. Cuny, Principal Investigator
Department of Computer and Information Science
University of Massachusetts, Amherst MA
Net Address: cuny@cs.umass.edu

1992

ONR Contract Number: N00014-89-J-1492

RESEARCH DESCRIPTION

1. Overview

Highly parallel architectures will be useful in meeting the demands of computationally intensive tasks only to the extent that it is possible to write efficient parallel software. The problems are enormous. The parallel programmer must simultaneously code for multiple processes, orchestrating their communication and synchronization; he must efficiently map logical processes onto disparate hardware configurations and schedule their execution. Further, he must debug – both for correctness and performance – in spite of a potentially overwhelming amount of relevant information and in the absence of reproducibility or consistent global states. If it is not possible to provide sophisticated programming support for these activities, it is unlikely that highly parallel computation will be generally available to either the scientific or the commercial communities.

In our research, we[1] have investigated aspects of parallel computation that are specific to massive parallelism. During most of the funding period, we focused on computations designed for MIMD, message-passing architectures, considering support for fine-grained parallelism in which large numbers of processes communicate frequently across regular interconnection structures. For these computations, we developed techniques for program

[1] I would like to acknowledge the contributions of the students who have worked on this project. Graduate students include Duane Bailey, Alfred Hough, Joydip Kundu, Bruce Leban, Kumar Varadaraju, and Qing Yu; undergraduates include Jim Ahrens, and Craig Loomis.

12

specification and visualization. More recently, we expanded our focus to include fine-grained SIMD computations and we developed optimizations for array convolutions.

2. Parallel Program Specification

The abstractions provided by a programming environment determine a programmer's effectiveness in implementing and debugging algorithms, yet few abstractions exist for massive parallelism. We began by considering the role of graph representations. Graphs provide a natural way of thinking about parallelism. Their explicit use can reduce the disparity between a programmer's conceptualization of his algorithm and its implementation, increase the homogeneity of process code and provide a basis for coherent graphical displays.

Existing programming environments, however, did not support the explicit use of graphs. Ideally, such support should provide for scalability, user-specific annotations, graph manipulations, and visualization. The most difficult of these is scalability but it is crucial since programmers typically implement and debug their programs in-the-small and then scale them for massive parallelism and even production programs may require rescaling to reflect problem size constraints or hardware availability. No existing tools provided this range of facilities.

We based our tool on a form of graph grammars – *Aggregate Rewriting (AR) Graph Grammars* [1,2,3] – that we had previously developed. AR grammars are particularly suited to descriptions of communication structures — structures that are connected and sparse with low degree, near symmetry, and low radius. They provide a flexible mechanism for the concise, graphical specification of entire graph families. Graph grammar formalisms are, however, quite foreign to most programmers.

As a result, we developed a grammar-based editor – called ParaGraph[2] – that provides a "friendly" user interface [4]. Using ParaGraph, the programmer begins by specifying the smallest member of his graph family. He then describes the set of transformations needed to convert that graph into the next larger family member and he develops a *script* to direct the order of

[2]Though the prefix "para" might suggest parallel (either because we use a parallel graph rewriting mechanism or because we apply our results to parallel programming), we interpret it to mean "*beyond*" (as in "paranormal"), emphasizing the fact that the editor supports the specification of not just single graphs, but entire graph families.

their application. The initial graph becomes the start graph of an underlying graph grammar, the transformations become its productions, and the script determines allowable derivation sequences. ParaGraph provides an interface to the basic AR mechanisms and it extends them in a number of ways — adding restricting predicates, edge inheritance, and graph composition — to make a more convenient tool for the programmer.

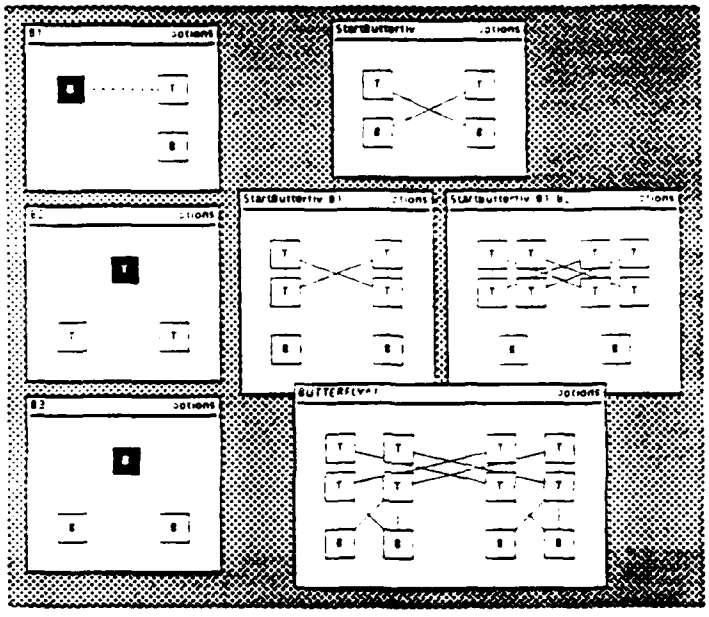


Figure 1: Butterfly grammar and the results of the first iteration of the script (`B1` through `B3`).

Figure 1 shows the sample definition of the family of butterfly graphs. The four node start graph (`StartButterfly`) is annotated by a single, user-defined attribute giving its `rank`. There are three transformations. The first (`B1`) begins a new rank of the butterfly by adding a row of nodes connected along the top level. We use dark, solid shading for the nodes being replaced and lighter shading for the nodes of the replacing graph. The application of this production is shown in `StartButterfly::B1`: it applies only to nodes in the top level because labels of matched graph instances must have `rank=0`. The second transformation (`B2`) makes a connected copy of the original host graph by rewriting nodes on the bottom two ranks (`StartButterfly::B1::B2`); here variations in shading indicate a partitioning of the edge inheritance function. The application of this production is

limited to the lower ranks of the butterfly by a *restricting predicate* (not shown). Restricting predicates are most often specified by example: the user selects a subset of nodes from a sample graph and heuristics are used to convert his selection into a generalized, closed form expression [5]. The third transformation (B3) makes a connected copy of the nodes in the top rank to complete the butterfly (BUTTERFLY^ 1). The appropriate script for this grammar is

```
StartButterfly:(B1 B2 B3)n.
```

BUTTERFLY^ 2 and BUTTERFLY^ 4, then, would be the 32 and 192 node butterflies respectively. The layout of the butterfly as shown was generated automatically. We provide both generation-time and post-generation layout heuristics.

Our motivation for the graph editor was to provide support for the explicit representation of graphs for use within a parallel programming environment. In our environment, we view a parallel program as an *annotated graph* [6]. Annotations might, for example, include code segments, run-time parameters, port associations, and compiler-time constants. Using ParaGraph, the programmer specifies a family of annotated graphs and then a specific instance of that family is generated and preprocessed into a form suitable for compilation. Currently we produce C code and channel declarations for execution on a multiprocessor simulator but extensions to other target architectures are straightforward. ParaGraph's output — in the form of both annotated graphs and underlying graph grammars — is accessible to all of our tools supporting program development.

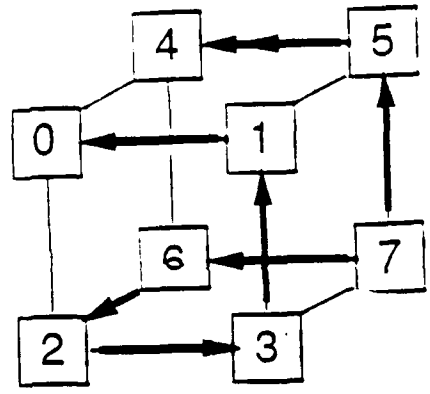
The use of ParaGraph still requires an understanding of graph grammar mechanisms. Programmers are comfortable with the concept of "growing" a large graph from a small graph, but they often find the subtleties of graph embedding mechanisms quite confusing. AR embedding mechanisms are particularly difficult because inheritance is determined by a partitioning of node and edge relations. Thus, we have begun design of a simplified interface for ParaGraph that removes these concerns from the programmer's domain [7]. It is based on the more familiar graph drawing operations of *copy* (which duplicates a subgraph) and *replace* (which replaces single nodes). The programmer uses these operations to draw a prototypical node replacement and the editor infers an underlying AR production. The simplification sacrifices some generality — for example, all inferred productions have single node

left-hand sides and uniform partitioning — but we have found that even sophisticated users seldom employ the full generality of AR grammars when defining graphs typical of parallel computation.

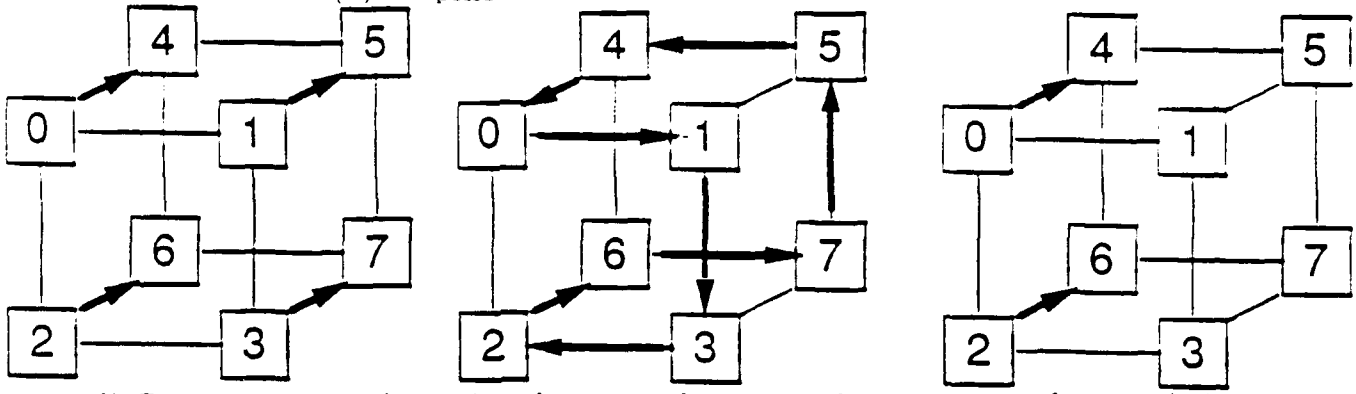
The editor was originally envisioned as a specification tool. Now that it has been integrated into our environment, however, we see it in a more central role, serving as a common graphical interface to other tools (debuggers, animators, mappers, *etc.*). Large graphs were not a problem during specification because the programmer only worked with small graphs which were automatically scaled just prior to compilation. Other support tools, however, will need access to the generated program graphs which may have thousands or even tens of thousands of nodes. Such graphs are prohibitively expensive to construct and extremely difficult to visualize. Thus, we also began to investigate the use of compact representations of graph derivations to provide efficient techniques for interrogating and visualizing large graphs without explicit construction [8]. Specifically, we are investigating derivation-based layout (layout and placement decisions are made locally as productions are applied), partial visualizations (abstractions of the graph structure are used without explicit rendering of all nodes), and lazy generation (limited construction of specified regions of the graph).

3. Parallel Program Animation

It is extremely difficult to understand the behavior of massively parallel systems: they have an overwhelming amount of potentially relevant information and often they do not have consistent global states or reproducible behavior. Visualization has been widely used as an aid, but standard visualization techniques do not address the fundamental problems of complexity and concurrency in parallel computations. Our approach combines event-based behavioral abstraction with animation: the programmer describes the intended behavior of his program with a high-level model that is then used to guide the animation of its actual behavior. We demonstrated this approach previously with the prototype of a pattern-oriented parallel debugger, called Belvedere [9,10]. Using Belvedere, we identified a fundamental problem in the visualization of abstract events: animations of concurrent, nonatomic events are often obscured because constituent subevents overlap in both time and space. This can be seen in Figure 2*a* where a snapshot of the animation of a simulated annealing of the traveling salesman problem reveals an incoherent jumble of communication events.



(*a*) Snapshot of low level communication events.



(*b*) Consecutive snapshots of *evaluate*, *synchronize*, and *swap* events after reordering.

Figure 2: Traveling Salesman.

In order to provide comprehensible animations of these events, we developed techniques for temporally reordering event streams with the goal of producing visually distinct animations of concurrent events [11]. In many cases, these reorderings — called *perspective views* — are accomplished without violating any program dependencies and thus result in equivalent, logically coherent animations. For the traveling salesman problem, we see in Figure 2*b* that the animation has been separated into three logically meaningful "abstract" steps: an *evaluate* step in which processes communicate across one of the cube dimensions to determine the value of proposed swaps; a *synchronize* step in which a token is passed around an embedded ring to insure that only nonconflicting swaps are accepted; and a *swap* step in which accepted swaps are made. It is possible to automatically separate these abstract events because there are no conflicting dependencies: all processes see the three steps in the same order and all interprocess communication happens within a step.

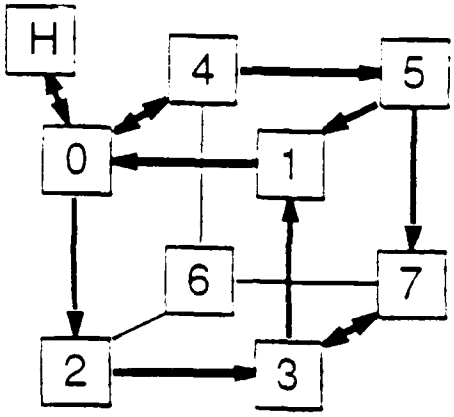
Not all abstract behaviors have this property.

Consider, for example, a program that issues queries to a database stored on a hypercube. The host issues queries which are routed through the cube to the appropriate node and then back again to the host. The user understands this system in terms of abstract queries that group all of the traffic in response to a single host query together. If we look at an animation of such queries as in Figure 3*a* we see that there may be several active queries at a time. Abstract queries are concurrent but, because they are not necessarily seen in the same order at each process, their dependencies can not be consistently separated. For such systems, we enable the user to construct partially consistent reorderings that preserve subsets of program dependencies. These *partial perspective views* provide only a limited view of the system behavior but they are easily constructed and they can be used in combinations to achieve a more comprehensive view. In dictionary search example we can separate the queries based on the order in which the host issues them to get the pictures in Figure 3*b*.

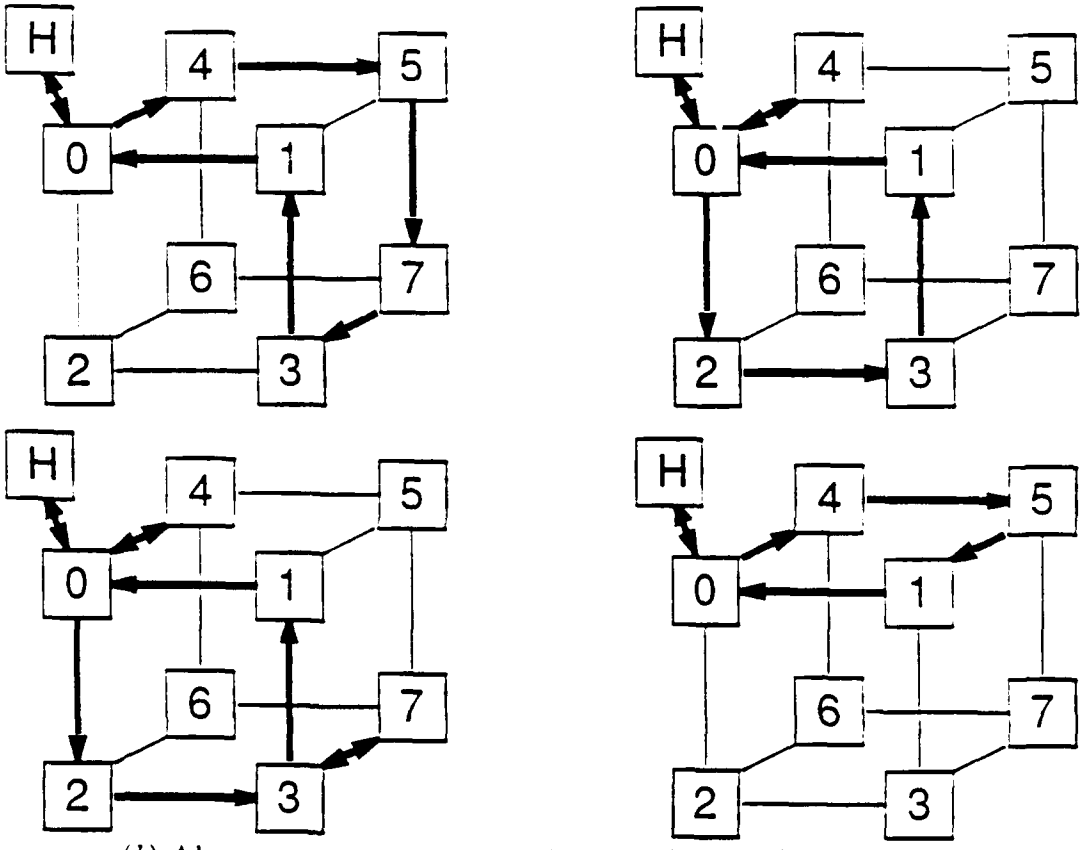
We have implemented partial perspectives within our debugger and found them to be quite useful in exposing bugs that had previously avoided detection. The techniques themselves are quite general. They can be applied to a number of other visualization tools as we demonstrated for the cases of process-time graphs and user-directed animations [11]. We have begun to evaluate them in a more general context by implementing perspective views within the Voyeur system [12]. Voyeur is a more conventional tool for displaying application-specific visualizations of parallel programs [13] and it provides a flexible experimental testbed for investigating a variety of issues such as

> Is user-defined, behavioral abstraction useful in a general animation system? Can we characterize the cases in which such animation is useful? What support does it require? Can we automatically generate visualizations of abstract events in more conventional animators? Is the manipulation of time meaningful in general animation systems? Is reordering only necessary in asynchronous systems? In the presence of abstraction? Can we provide visual cues to dependency violations?

We have only very preliminary results from this work but we expect that



(*a*) Concurrent queries (no reordering).



(*b*) Abstract query events reordered with partial perspective.

Figure 3: Dictionary Search.

further investigation will clarify the role of behavioral abstraction and time manipulations in understanding complex behaviors.

4. Other Work: A Convolution Optimizer for SIMD Programs

Communication overhead can easily offset performance increases due to massive parallelism. The overhead is particularly significant for fine-grained, SIMD architectures since relatively little computation is performed between successive communications and all processes are delayed while communication completes. We have developed code optimizations for SIMD architectures that reduce communication costs for array convolutions, an important class of array manipulations [14].

Our work began as an adaptation of algebraic optimization techniques developed by Fisher and Highnam [15]. In adapting their heuristics for the Connection Machine, we attempted to address their reliance on a directional algebra that applied only to grids. They assumed the machine architecture was a grid of dimension less than or equal to that of the input array which menat that they are unable to fully exploit the interconnectivity of an architecture such as the Connection Machine. Our heuristics use graph theoretic techniques and they are more general: they eliminate restrictions on the architecture and permit input structures with other topologies.

References

1 Duane A. Bailey and Janice E. Cuny, "Graph Grammar Based Specification of Interconnection Structures for Massively Parallel Computation," *Proceedings Third International Workshop on Graph Grammars, Lecture Notes on Computer Science*, pp. 73-85 (1987).

2 Duane A. Bailey and Janice E. Cuny, "An Approach to Programming Process Interconnection Structures: Aggregate Rewriting Graph Grammars," *Parallel Architectures and Languages Europe, Lecture Notes in Computer Science 259*, J.W. de Bakker, A.J. Nijman and P.C. Treleaven (eds.), Springer-Verlag, pp.112-123 (June 1987).

3 Duane A. Bailey, *Specifying Communication for Massively Parallel Ensemble Machines*. Ph.D. Thesis, COINS Department, University of Massachusetts (1988).

4 Duane A. Bailey, Janice E. Cuny, and Craig P. Loomis. "ParaGraph: Graph Editor Support for Parallel Programming Environments." *International Journal of Parallel Programming* 19(2), pp. 75-110 (April 1990).

5 Qing Yu and Janice E. Cuny, "Support for Subgraph Identification in a Parallel Programming Environment." *Proceedings of the First Annual IEEE Symposium on Distributed and Parallel Processing*, Dallas, TX, pp. 196-197 (May 1989).

6 Duane A. Bailey and Janice E. Cuny, "Visual Extensions to Parallel Programming Languages" in *Languages and Compilers for Parallel Computing*, David Gelernter, Alexandru Nicolau, and David Padua (eds.), The MIT Press, Cambridge Massachusetts, Chapter 2, pp. 17-36 (1990).

7 Charles D. Fisher, "Approaches to Specifying Aggregate Rewriting Graph Grammar Productions," M.S. Thesis, COINS Department, University of Massachusetts (1990).

8 Duane A. Bailey, Janice E. Cuny, and Charles D. Fisher, "Programming with *Very* Large Graphs." Accepted for publication, Fourth International Workshop on Graph Grammars and their Applications to Computer Science.

9 Alfred A. Hough and Janice E. Cuny, "Belvedere: Prototype of a Pattern-Oriented Debugger for Highly Parallel Computation." *Proceedings of the 1987 International Conference on Parallel Processing*, pp. 735-738 (1987).

10 Alfred A. Hough and Janice E. Cuny, "Initial Experiences with a Pattern-Oriented Debugger." *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pp. 195-205 (May 1988). Also appeared *SIGPLAN Notices* 24(1), pp. 195-205 (January 1989).

11 Alfred A. Hough and Janice E. Cuny, "Perspective Views: A Technique for Enchancing Visualizations of Parallel Programs." *Proceedings of the 1990 International Conference on Parallel Processing*, pp. II 124-132 (August 1990). Long version COINS Technical Report 90-02.

12 Nandakumar Varadaraju. Interfacing Belvedere with Voyeur. Master's Thesis. COINS Department. University of Massachusetts (June 1991).

13 David Socha and Mary L. Bailey and David Notkin. "Voyeur: Graphical Views of Parallel Programs", *SIGPLAN Workshop on Parallel and Distributed Debugging*, pp. 206-215 (1988).

14 Joydip Kundu and Janice E. Cuny, Optimizations of Array Convolutions for SIMD Architectures. COINS Technical Report 91-65. University of Massachusetts (September 1991).

15 Allan L. Fisher and Peter T. Highnam. "Communication and Code Optimization in SIMD Programs" . *Proceedings of the 1989 International Conference on Parallel Processing*, pp. 84-88 (1989).

PUBLICATIONS/REPORTS

Alfred A. Hough, *Debugging Parallel Programs Using Abstract Visualizations.* PhD Thesis, COINS Department, University of Massachusetts (1991).

Joydip Kundu and Janice E. Cuny, Optimizations of Array Convolutions for SIMD Architectures. COINS Technical Report 91-65, University of Massachusetts (September 1991).

Nandakumar Varadaraju. *Interfacing Belvedere with Voyeur.* Master's Thesis, COINS Department, University of Massachusetts (June 1991).

Duane A. Bailey, Janice E. Cuny, and Charles D. Fisher. "Programming with *Very* Large Graphs." Accepted for publication, Fourth International Workshop on Graph Grammars and their Applications to Computer Science.

Duane A. Bailey, Janice E. Cuny, and Craig P. Loomis. "ParaGraph: Graph Editor Support for Parallel Programming Environments." *International Journal of Parallel Programming* 19(2), pp. 75-110 (April 1990).

Duane A. Bailey and Janice E. Cuny, "Visual Extensions to Parallel Programming Languages" in *Languages and Compilers for Parallel Computing*, David Gelernter, Alexandru Nicolau, and David Padua (eds.), The MIT Press, Cambridge Massachusetts, Chapter 2, pp. 17-36 (1990).

Alfred A. Hough and Janice E. Cuny, "Perspective Views: A Technique for Enchancing Visualizations of Parallel Programs." *Proceedings of the 1990 International Conference on Parallel Processing*, pp. II 124-132 (August 1990). Long version COINS Technical Report 90-02.

Nandakumar Varadaraju, The ParaGraph Tutorial. COINS Technical Report 90-51 (June 1990).

David K. Black, ParaGraph: The User's Manual. COINS Technical Report 90-35 (May 1990).

Alfred A. Hough and Janice E. Cuny, Perspective Views: A Technique for Enchancing Visualizations of Parallel Programs (Long Version). COINS Technical Report 90-02 (1990).

Charles D. Fisher, "Approaches to Specifying Aggregate Rewriting Graph Grammar Productions," M.S. Thesis, COINS Department, University of Massachusetts (1990).

Qing Yu and Janice E. Cuny, "Support for Subgraph Identification in a Parallel Programming Environment." *Proceedings of the First Annual IEEE Symposium on Distributed and Parallel Processing*, Dallas, TX, pp. 196-197 (May 1989).

Mark Gisi, Janice E. Cuny and Duane A. Bailey, "Canister Communication as a Vehicle for Parallel Debugging," *Proceedings of the First Annual IEEE Symposium on Distributed and Parallel Processing*, Dallas, TX, pp. 198-199 (May 1989).

Duane A. Bailey and Janice E. Cuny, "Cannister Communication in Parallel Programs." COINS Technical Report 88-42 (October 1988).

HONORS

Janice E. Cuny, IEEE Distinguished Visitor, 1990-1992

Janice E. Cuny, NSF Faculty Award for Women 1991